SSJ User's Guide

Package **`simexp`**

Tools for Simulation Experiments

Version: September 8, 2014

Contents

Overview of package `simexp`

Provides some classes to manage simulation experiments. Let $\bar{\mathbf{X}}_n$ be an average of random vectors:

$$\bar{\mathbf{X}}_n = \frac{1}{n}\sum_{r=0}^{n-1}\mathbf{X}_r,$$

where $\mathbf{X}_r$ is the rth observation obtained during an experiment. Assuming that the $\mathbf{X}_r$'s are i.i.d., $\{\bar{\mathbf{X}}_n, n \geq 0\}$ is a sequence of vectors in $\mathbb{R}^d$ converging to a vector $\boldsymbol{\mu} = E[\mathbf{X}_r]$ when $n \to \infty$. We use simulation to estimate this $\boldsymbol{\mu}$.

The simplest way to generate the sample $(\mathbf{X}_0, \ldots, \mathbf{X}_{n-1})$ is by simulating the same system n times, independently. In that setting, r becomes the index of a replication. In general, a simulation generates n copies of $\mathbf{X}_r$ to compute $\bar{\mathbf{X}}_n$, in order to estimate $\boldsymbol{\mu}$. We may also be interested in some sample covariances of components of $\mathbf{X}_r$, for computing confidence intervals on functions of $\boldsymbol{\mu}$. The most common functions return a single component of $\boldsymbol{\mu}$, or a ratio of two components.

For example, when simulating a $M/M/1$ queue, we may be able to get the total waiting time W for all customers, the number N of served customers, and the integral of the queue size over simulation time $Q(T) = \int_0^T q(t)\ dt$, where $q(t)$ is the queue size at time t. In this case, $\mathbf{X}_r = (W_r, N_r, Q_r(T))$, and $\boldsymbol{\mu} = (E[W], E[N], E[Q(T)])$. Two interesting functions of $\boldsymbol{\mu}$ are the expected waiting time per customer $E[W]/E[N]$, and the time-average queue size $E[Q(T)]/T$. The functions are simply evaluated at $\bar{\mathbf{X}}_n$ to estimate the latter quantities.

The number of observations n is usually constant, but it may also be random if *sequential sampling* is used. With this scheme, after n_0 observations are available, an error check is performed to determine if simulation should continue. For example, this check may evaluate the relative error of an estimated performance measure by dividing the half-width of a computed confidence interval by the point estimator, and require additional observations if this error is too high. The procedure is repeated until the stopping conditions are verified, or a maximal number of observations is obtained. However, because the sample size n is random, the estimator $\bar{\mathbf{X}}_n$ is biased when using sequential sampling.

The vector $\mathbf{X}_r$ is usually computed by summing costs incurred for various events during a part of the experiment. These costs can be waiting times, number of items, etc., not necessarily money. The computed sums may then be processed in a way depending of the simulated horizon and model to get the required values. If the horizon is finite, simulation stops after a finite time T, or a finite number N of events, and is usually repeated n times independently. Then, for replication r,

$$\mathbf{X}_r = \sum_{k=0}^{N-1}\mathbf{C}_{k,r}$$

or

$$\mathbf{X}_r = \sum_{k=0}^{N_r(T)-1}\mathbf{C}_{k,r}$$

where $\mathbf{C}_{k,r}$ is the cost of the kth event during replication r, and $N_r(T)$ is the total number of events occurring during the time interval $[0, T]$. This estimates the total expected cost over the horizon.

When the horizon is infinite, a single replication is usually simulated, and the cost per time unit or per event is computed in the long run in order to estimate

$$\boldsymbol{\mu} = \lim_{N\to\infty} \left(\frac{1}{N} E\left[\sum_{k=0}^{N-1} \mathbf{C}_k \right] \right) = \lim_{N\to\infty} \left(\frac{1}{N} \sum_{k=0}^{N-1} \mathbf{C}_k \right)$$

or

$$\boldsymbol{\mu} = \lim_{T\to\infty} \left(\frac{1}{T} E\left[\sum_{k=0}^{N(T)-1} \mathbf{C}_k \right] \right) = \lim_{T\to\infty} \left(\frac{1}{T} \sum_{k=0}^{N(T)-1} \mathbf{C}_k \right).$$

Any estimator of these previous quantities is biased, because the horizon must be truncated, and the model does not necessarily start in steady state. A single long replication is simulated to reduce the bias, and the first events are dropped (warmup).

However, with a single long replication, computing sample covariances and confidence intervals is more difficult. A simple technique to overcome this problem is *batch means* [1], which divides the truncated horizon into successive intervals called *batches*. More specifically, let T_0 be the time at which the warmup ends. We divide the horizon $[T_0, T]$ in n batches starting at times $T_0 < \cdots < T_{n-1}$, and the last batch ends at $T_n = T > T_{n-1}$. Then, for batch r,

$$\mathbf{X}_r = \sum_{k=N(T_r)}^{N(T_{r+1})-1} \mathbf{C}_k,$$

where $r = 0, \ldots, n-1$. Batches may have a fixed duration in simulation time units, contain a fixed number of events, etc. The beginning of each batch can even correspond to a *regeneration point*, i.e., a simulation time at which the state and behavior of the model does not depend on the past. In the latter case, each batch corresponds to a *regenerative cycle*. In all these cases, $\boldsymbol{\mu}$ is estimated by

$$\frac{\sum_{r=0}^{n-1} \mathbf{X}_r}{\sum_{r=0}^{n-1} (T_{r+1} - T_r)} = \frac{\sum_{r=0}^{n-1} \mathbf{X}_r}{T - T_0}.$$

The simplest way for estimating covariances when all batches have the same length is to consider the $\mathbf{X}_r$'s as i.i.d. random vectors, and use the same techniques as with independent replications. For regenerative cycles, confidence intervals must be computed on ratios of means. The class `FunctionOfMultipleMeansTally` can be used for this.

This package provides helper classes to facilitate management of a complex simulation experiment. A base class called `SimExp` contains methods to initialize lists of statistical probes and to help in sequential sampling. The subclass `RepSim` is used for simulating independent replications of a given model on a finite horizon. The subclass `BatchMeansSim` can be used for simulating a stationary model using the batch means technique.

SimExp

Represents a framework for performing experiments using simulation. This class defines an abstract `simulate` method that should implement the simulation logic. It also provides utility methods to estimate the required number of additional observations that would be necessary for an estimator to reach a given precision, for sequential sampling.

This class is the base class of `BatchMeansSim` and `RepSim` implementing the logic for a simulation on infinite and finite horizon, respectively.

```
package umontreal.iro.lecuyer.simexp;
```

```
public abstract class SimExp
```

```
protected boolean simulating
```

Determines if the simulation is in progress.

```
protected SimExp ()
```

Constructs a new object for performing experiments using the default simulator returned by `Simulator.getDefaultSimulator()`.

```
protected SimExp (Simulator sim)
```

Constructs a new object performing experiments using the given simulator `sim`.

```
public final Simulator simulator ()
```

Returns the simulator linked to this experiment object.

```
public final void setSimulator (Simulator sim)
```

Sets the simulator associated with this experiment to `sim`. This method should not be called while this object is simulating.

```
public boolean isSimulating()
```

Determines if the simulation is in progress.

```
public abstract void simulate()
```

Performs an experiment whose logic depends on the used subclass. Before starting the simulation, this method should set `simulating` to `true`, and reset it to `false` after the simulation is done. It is recommended to reset `simulating` to `false` inside a `finally` clause to prevent the indicator from remaining `true` in the case of error during simulation.

```
public static int getRequiredNewObservations (StatProbe[] a,
                                               double targetError,
                                               double level)
```

Returns the approximate number of additional observations required to reach a relative error smaller than or equal to `targetError` for each tally in the array `a` when confidence intervals are computed with confidence level `level`. For each statistical collector in the given array, a confidence interval is computed independently of the other collectors, and an error check is performed by `getRequiredNewObservations` to determine the required number of additional observations. The method returns the maximal number of required observations.

```
public static int getRequiredNewObservations (
                                  Iterable<? extends StatProbe> it,
                                              double targetError,
                                              double level)
```

Returns the approximate number of additional observations required to reach a relative error smaller than or equal to `targetError` for each tally enumerated by `it` when confidence intervals are computed with confidence level `level`. For each statistical collector returned by the iterator obtained from `it`, a confidence interval is computed independently of the other collectors, and an error check is performed by `getRequiredNewObservations` to determine the required number of additional observations. The method returns the maximal number of required observations.

```
public static int getRequiredNewObservations (StatProbe probe,
                                              double targetError,
                                              double level)
```

Calls `getRequiredNewObservations` with the average, confidence interval radius, and number of observations given by the statistical probe `probe`. This method always returns 0 if the probe is not a tally. For a `Tally`, the confidence interval is computed using `confidenceIntervalStudent`. For a `FunctionOfMultipleMeansTally`, it is computed using `confidenceIntervalDelta`.

```
public static int getRequiredNewObservationsTally (Tally ta,
                                              double targetError,
                                              double level)
```

Calls `getRequiredNewObservations` with the average, confidence interval radius, and number of observations given by the tally `ta`. The confidence interval is computed using `confidenceIntervalStudent`.

```
public static int getRequiredNewObservationsTally (
                                   FunctionOfMultipleMeansTally fmmt,
                                              double targetError,
                                              double level)
```

Calls `getRequiredNewObservations` with the average, confidence interval radius, and number of observations given by the function of multiple means `fmmt`. The confidence interval is computed using `confidenceIntervalDelta`.

```
public static int getRequiredNewObservations (double center,
                                              double radius,
                                              int numberObs,
                                              double targetError)
```

Returns the approximate number of additional observations needed for the point estimator $\bar{X}_n$ = `center`, computed using n = `numberObs` observations and with a confidence interval having radius $\delta_n/\sqrt{n}$ = `radius`, to have a relative error less than or equal to ϵ = `targetError`. It is assumed that $\bar{X}_n$ is an estimator of a mean μ, n is the number of observations `numberObs`, and that $\delta_n/\sqrt{n} \to 0$ when $n \to \infty$.

If n is less than 1, this method returns 0. Otherwise, the relative error given by $\delta_n/|\sqrt{n}\bar{X}_n|$ should be smaller than or equal to ϵ. If the inequality is true, this returns 0. Otherwise, the minimal n^* for which this inequality holds is approximated as follows. The target radius is given by $\delta^* = \epsilon|\mu|$, which is approximated by $\epsilon|\bar{X}_n| < \delta_n/\sqrt{n}$. The method must select n^* for which $\delta_{n^*}/\sqrt{n^*} \le \delta^*$, which will be approximately true if $\delta_{n^*}/\sqrt{n^*} \le \epsilon|\bar{X}_n|$. Therefore,

$$n^* \ge (\delta_{n^*}/(\epsilon|\bar{X}_n|))^2 \approx (\delta_n/(\epsilon|\bar{X}_n|))^2.$$

The method returns $\text{round}((\delta_n\sqrt{n}/(\epsilon|\bar{X}_n|))^2) - n$ where $\text{round}(\cdot)$ rounds its argument to the nearest integer.

RepSim

Performs a simulation experiment on a finite horizon, using a certain number of independent runs or replications. During each run r, a complete simulation is executed, and the vector $\mathbf{X}_r$ is generated. If simulation runs are independent, and the same system is simulated during each run, after n runs are performed, a sample $(\mathbf{X}_0, \ldots, \mathbf{X}_{n-1})$ is obtained.

For such a simulation to be implemented, this class must be extended to override the required methods: `initReplicationProbes` to initialize the statistical probes collecting $\mathbf{X}_r's$, `initReplication` to initialize the simulation model at the beginning of each replication, and `addReplicationObs` to add $\mathbf{X}_r$ to the statistical probes.

```
package umontreal.iro.lecuyer.simexp;

public abstract class RepSim extends SimExp
```

Constructors

`public RepSim (int minReps)`

Constructs a new replications-based simulator with a minimal number of runs, `minReps`, and no maximal number of runs.

`public RepSim (int minReps, int maxReps)`

Constructs a new replications-based simulator with a minimal number of runs `minReps`, and a maximal number of runs `maxReps`. This maximum is used to avoid too long simulations when using sequential sampling.

`public RepSim (Simulator sim, int minReps)`

Equivalent to the first constructor, with the given simulator `sim`.

`public RepSim (Simulator sim, int minReps, int maxReps)`

Equivalent to the second constructor, with the given simulator `sim`.

Methods

`public int getMinReplications()`

Returns the minimal number of replications to be simulated before an error check.

`public void setMinReplications (int minReps)`

Sets the minimal number of replications required before an error check to `minReps`. This also updates the maximal number of replications if this maximum is smaller than the new minimum. This will take effect only at the next call to `simulate`.

`public int getMaxReplications()`

Returns the maximal number of replications to be simulated before an error check. By default, this is set to `MAX_VALUE`, which is equivalent to infinity in practice.

`public void setMaxReplications (int maxReps)`

Sets the maximal number of replications required before an error check to `maxReps`. This will take effect only at the next call to `simulate`.

`public int getTargetReplications()`

Returns the actual target number of replications to be simulated before an error check. By default, this is initialized to the minimal number of replications, and is increased if new replications are needed. However, it is not decreased by default, even upon a new call to `simulate`.

`public void setTargetReplications (int targetReps)`

Sets the target number of simulated replications before an error check to `targetReps`. The value of `targetReps` must not be smaller than the minimal number of replications returned by `getMinReplications`, or greater than the maximal number of replications returned by `getMaxReplications`.

`public int getCompletedReplications()`

Returns the total number of completed replications for the current experiment.

`public abstract void initReplicationProbes()`

Initializes any statistical collector used to collect values for replications.

`public void performReplication (int r)`

Contains the necessary logic to perform the `r`th replication of the simulation. By default, the method calls `init` to clear the event list, and uses `initReplication` to initialize the model. It then calls `start` to start the simulation, calls `replicationDone` to increment the number of completed replications, and `addReplicationObs` to add observations to statistical probes.

`protected void replicationDone()`

Increments by one the number of completed replications. This is used by `performReplication`.

`public abstract void initReplication (int r)`

Initializes the simulation model for a new replication `r`. This method should reset any counter and model state, and schedule needed events. After the method returns, the model should be ready for calling `start`. This method is called just after the simulator is initialized.

`public abstract void addReplicationObs (int r)`

Adds statistical observations for the replication `r`. This method is called just after the replication `r` is simulated.

`public int getRequiredNewReplications()`

Returns the approximate number of additional replications to meet an experiment-specific stopping criterion. This is called after `getTargetReplications` replications are simulated. Since sequential sampling is not used by default, the default implementation returns 0, which stops the simulation after `getTargetReplications` replications.

`public void init()`

Initializes this simulator for a new experiment. This method resets the number of completed replications to 0, and calls `initReplicationProbes` to initialize statistical probes. This method is called by `simulate`.

`public void adjustTargetReplications (int numNewReplications)`

Adjusts the target number of replications to simulate `numNewReplications` additional replications. This method increases the target number of replications by `numNewReplications`, and sets the target number of replications to `getMaxReplications` if the new target exceeds the maximal number of replications. This is called by `simulate` for sequential sampling.

`public void simulate()`

Simulates several independent simulation replications of a system. When this method is called, the method `init` is called to initialize the system, and `getTargetReplications` replications are simulated by using `performReplication`. When the target number of replications is simulated, the stopping condition is checked using `getRequiredNewReplications`, and the target number of replications is adjusted using `adjustTargetReplications`. Additional replications are simulated until the method `getRequiredNewReplications` returns 0, or `getMaxReplications` replications are simulated.

BatchMeansSim

Performs a simulation experiment on an infinite horizon, for estimating steady-state performance measures, using batch means. Batches are delimited using a user-specified condition such as a fixed duration in simulation time units (the default), the number of occurrences of an event such as the arrival of a customer, a regenerative cycle, etc. After the condition for batch termination is defined, the *batch size* can be set. This size can be, depending on how batches are delimited, a time duration, a number of events, or 1 for regenerative cycles. The *batch length* is defined to be the duration of a batch, in simulation time units, independently of how batches are defined. By default, the batch size and the batch length are equivalent (and constant), but they may differ if the condition for batch termination is changed.

A warmup period is usually simulated to reduce the bias induced by the initial state of the system. During the warmup, the system runs without any observation being collected. By default, this period has a duration fixed in simulation time units, but this can also be changed.

After the warmup is over, events are counted as follows. The simulation model defines some counters being updated when events occur and reset at the beginning of each batch. At the end of a batch, the value of these counters are used to generate a random vector $\mathbf{V}_j$ before the counters are reset. Alternatively, a simulation may compute and update the $\mathbf{V}_j$'s directly, without using intermediate counters. A set of data structures is needed to collect and store these $\mathbf{V}_j$'s, the simplest option being a set of `TallyStore` instances. This generates values for m *real batches*. The sample $(\mathbf{X}_0, \ldots, \mathbf{X}_{n-1})$ is then obtained from $(\mathbf{V}_0, \ldots, \mathbf{V}_{m-1})$, so a second set of data structures is needed to collect the $\mathbf{X}_r$'s, the simplest being a set of `Tally` instances. The most straightforward way to estimate covariances on components of $\mathbf{X}_r$ is by considering the vectors $\mathbf{X}_r$ i.i.d., which is not true in general. However, by choosing a sufficiently large simulation time and batch lengths, the correlation between batches can be reduced, and the $\mathbf{X}_r$'s are approximately i.i.d. and normally distributed. If batches correspond to regenerative cycles, the $\mathbf{X}_r$ are then truly i.i.d., but only approximately normally distributed. Confidence intervals on functions of $\boldsymbol{\mu}$ can be approximated using the central limit theorem, as with independent replications, or using the delta theorem for functions of multiple means or when batches have different lengths.

The sample size corresponding to the number of simulated batches is always fixed when sequential sampling is not used; we have $n = m$ and $\mathbf{X}_r = \mathbf{V}_r$ for $r = 0, \ldots, n-1$. However, when sequential sampling is used, the sample size can be random or fixed. If the sample size is random, we still have $\mathbf{X}_r = \mathbf{V}_r$. When using this mode, the batch size must be chosen carefully to reduce correlation between batches.

On the other hand, if the sample size n is required to be fixed while sequential sampling is used, *batch aggregation* must be activated to have *effective batches* with random lengths. When aggregation is enabled, real batches are simulated as usual, but the obtained values $\mathbf{V}_j$ are regrouped (or aggregated) to form effective batches. The number of real batches must be $m = h * n$ to get a sample of size n, and h can be any integer greater than or equal to 1. In this case,

$$\mathbf{X}_r = \sum_{k=0}^{h-1} \mathbf{V}_{hr+k}.$$

The size of the effective batches increases while the sample size remains fixed because the group size h increases with simulation length. When aggregation is used, the effective batches become longer while simulation time increases, and the correlation between effective batches should decrease with simulation time. However, the increment to the target number of batches must always be a multiple of the sample size, which can lead to useless simulation if the sample size is large.

This class must be extended to implement a batch means simulator, and the appropriate methods must be defined or overridden. This class uses a simulation event to stop the simulation at the end of the warmup period and batches. One can use this event for fixed-duration warmup and batches, or schedule their own events which call `Sim.stop` to end warmup or batches. To change how the warmup period is terminated, one must override the method `warmup`. For the batch termination condition to be redefined, `simulateBatch` must be overridden.

One must implement `initSimulation` to initialize the simulated model before the warmup, `initBatchStat` to reset the model-specific counters used to compute the $\mathbf{V}_j$'s, `initRealBatchProbes` and `addRealBatchObs` to initialize statistical probes and add observations for real batches, `initEffectiveBatchProbes` and `addEffectiveBatchObs` to initialize statistical probes and add observations for effective batches.

The moment the latter methods are called depends on the status of batch aggregation: when aggregation is turned ON, before any error check or the end of the simulation, `initEffectiveBatchProbes` is called once before `addEffectiveBatchObs` is called n successive times, with different parameters. When aggregation is turned OFF, then method `initEffectiveBatchProbes` is called after the warmup is over, and `addEffectiveBatchObs` is called each time a batch ends, after `addRealBatchObs` is called.

If sequential sampling is used, `getRequiredNewBatches` must be overridden to implement error checking. In some particular situations, the user may also need to override `allocateCapacity` and `regroupRealBatches`.

```
package umontreal.iro.lecuyer.simexp;


public abstract class BatchMeansSim extends SimExp
```

Constructors

```
public BatchMeansSim (int minBatches, double batchSize,
                      double warmupTime)
```

Constructs a new batch means simulator using at least `minBatches` batches with size `batchSize`, with a warmup period of duration `warmupTime`. By default, batch aggregation and batch lengths keeping are turned off, and the maximal number of batches is infinite.

```
public BatchMeansSim (int minBatches, int maxBatches, double batchSize,
                      double warmupTime)
```

Constructs a batch means simulator with a maximum of `maxBatches` batches to avoid excessive memory usage and too long simulations when using sequential sampling. See `BatchMeansSim` for more information about the other parameters.

```
public BatchMeansSim (Simulator sim, int minBatches, double batchSize,
                      double warmupTime)
```

Equivalent to the first constructor, with a user-defined simulator `sim`.

```
public BatchMeansSim (Simulator sim, int minBatches, int maxBatches,
                      double batchSize, double warmupTime)
```

Equivalent to the second constructor, with a user-defined simulator `sim`.

Methods

`public boolean getBatchAggregation()`

Returns `true` if the aggregation of batches is turned ON. If `getRequiredNewBatches` always returns 0 (the default), the aggregation has no effect since the number of batches is not random. By default, batch aggregation is turned OFF.

`public void setBatchAggregation (boolean a)`

Sets the batch aggregation indicator to `a`. This should not be called during an experiment.

`public boolean getBatchLengthsKeeping()`

Indicates that the length, in simulation time units, of each real batch has to be kept. By default, this is set to `false`. When batch aggregation is turned ON, the batch lengths are always kept.

`public void setBatchLengthsKeeping (boolean b)`

Sets the batch lengths keeping indicator to `b`. This has no impact if batch aggregation is turned ON.

`public int getMinBatches()`

Returns the minimal number of batches required for estimating the steady-state performance measures of interest. If aggregation is turned ON, this is also the final number of effective batches, i.e., the sample size.

`public void setMinBatches (int minBatches)`

Sets the minimal number of batches to `minBatches`.

`public int getMaxBatches()`

Returns M, the maximal number of batches to be used for estimating the steady-state performance measures of interest. This is used to prevent the number of batches from growing indefinitely when using sequential sampling. By default, this is set to `MAX_VALUE`, which is equivalent to infinity in practice.

`public void setMaxBatches (int maxBatches)`

Sets the maximal number of batches to `maxBatches`.

`public double getBatchSize()`

Returns the current batch size as defined for this simulator. By default, this is a duration in simulation time units. Depending on the batch termination condition, which can be changed by overriding `simulateBatch`, it can be the number of occurrences of an event, or 1 for regenerative cycles.

`public void setBatchSize (double batchSize)`

Sets the batch size to `batchSize`.

`public double getWarmupTime()`

Returns the duration of the warmup period for the simulation. By default, this duration is expressed in simulation time units, but this can be changed by overriding `warmup`.

`public void setWarmupTime (double warmupTime)`

Sets the warmup time to `warmupTime`. If this method is called while a simulation is in progress, the new time will affect the next simulation only.

`public double getBatchFraction()`

Returns the remaining fraction of batch to be simulated. This method is called when scheduling the end of the next batch during a simulation. Sometimes, it can be necessary to increase the batch size to avoid excessive memory usage. In this case, stored real batches (the $\mathbf{V}_j$'s) have to be regrouped to use less memory, and the last stored $\mathbf{V}_j$ may represent an incomplete batch with respect to the new batch size. This method returns the fraction of batch, with respect to the new batch size, remaining to be simulated before a new batch starts. If regrouping is not used, this always returns 1. The returned value is always greater than 0 and smaller than or equal to 1. For example, with a fixed time batch size `s`, the next end of batch will be scheduled in `s*getBatchFraction()` simulation time units. This method returns values different from 1 only if one overrides `allocateCapacity`, and `regroupRealBatches`.

`public double getBatchSizeMultiplier()`

Returns the batch size multiplier after the simulation of a new batch. This is called when scheduling the end of a new batch, to multiply the batch size after `regroupRealBatches` has regrouped real batches. This can return any value greater than 0, or 1 if the size is unchanged (the most common case) or if aggregation is not used. This method returns values different from 1 only if one overrides `allocateCapacity`, and `regroupRealBatches`.

`public int getTargetBatches()`

Returns the target number of simulated real batches at the next time the stopping condition is checked. By default, this number is set to the minimal number of batches and is increased if the stopping condition check requires new simulated batches. This target number of batches is not decreased automatically, even upon a new call to `simulate`.

`public void setTargetBatches (int targetBatches)`

Sets the target number of simulated batches before an error check or the end of the simulation to `targetBatches`.

`public int getCompletedRealBatches()`

Returns the number of completed real batches since the beginning of the run.

`public int getDroppedRealBatches()`

Returns the number of real batches dropped. When using sequential sampling, the target number of batches can become very high, resulting in not enough memory available to store the real batches. One simple heuristic to address this issue is to drop the first real batches, and increase the batch length. This method gives the number of real batches which have been dropped to save memory.

`public void dropFirstRealBatches (int n)`

Drops the `n` first real batches to save memory.

`public int getBatch (double time)`

Returns the real batch corresponding to simulation time `time` when batch lengths are kept. If batch lengths are not kept, or if the given time corresponds to the warmup period, this method returns a negative value.

`public boolean isWarmupDone()`

Determines if the warmup period for the simulation is over.

`public int getNumAggregates()`

Returns h, the number of real batches contained into an effective batch. If aggregation is turned OFF, this always returns 1 as soon as at least one batch is simulated. Otherwise, this returns a number greater than or equal to 1 as soon as `addEffectiveBatchObs` is called.

`public double getRealBatchLength (int batch)`

Returns the length, in simulation time units, of the real batch `batch`. If batch lengths are not kept, this method can return the length of the last batch only.

`public double getRealBatchStartingTime (int batch)`

Returns the starting simulation time of batch `batch`.

`public double getRealBatchEndingTime (int batch)`

Returns the ending simulation time of batch `batch`.

`public void allocateCapacity (int capacity)`

Allocates the necessary memory for storing `capacity` real batches. When using sequential sampling, if the variance of estimators is high, many additional batches may be needed to reach the target precision. To avoid memory problems after a long simulation time, this method can preallocate the necessary memory. The method must ensure that the data structures used to store the $\mathbf{V}_j$'s can contain `capacity` real batches. This is done by recreating arrays or resizing data structures. By default, this method throws an `UnsupportedOperationException`. `regroupRealBatches` must be implemented if this method does not throw this exception.

`public void regroupRealBatches (int x)`

Regroups real batches `x` by `x`. When memory is low, the simulator can try to regroup real batches and increase the batch size consequently. This is partly done by this method. The user must override it and modify the internal data structures storing the $\mathbf{V}_j$'s in order to regroup elements. The number of real batches m becomes $m' = \lfloor m/x \rfloor$. After this method returns, each new $\mathbf{V}_j$ should contain $\mathbf{V}_j = \sum_{l=0}^{x-1} \mathbf{V}_{jx+l}$ for $j = 0, \ldots, m' - 1$, $\mathbf{V}_{m'} = \sum_{l=0}^{(m \bmod x)-1} \mathbf{V}_{m'x+l}$, and $\mathbf{V}_j = 0$ for $j = m' + 1, \ldots, m - 1$. Some static methods called `regroupElements` are provided by this class to help the user with this. By default, this method throws an `UnsupportedOperationException`, disabling this functionality which is not always needed.

`public abstract void initSimulation()`

Initializes the simulator for a new run. This is called by the `init` method after `init` is called.

`public abstract void initBatchStat()`

Resets the counters used for computing observations during the simulation at the beginning of a new batch.

`public abstract void initRealBatchProbes()`

Initializes any statistical collector for real batches. This is called at the end of the warmup period.

`public abstract void initEffectiveBatchProbes()`

Initializes any statistical collector for effective batches. This is called at every stopping condition check when aggregation is ON, or at the end of the warmup period when it is OFF.

`public abstract void addRealBatchObs()`

Collects values of a $\mathbf{V}_j$ vector concerning the last simulated real batch. This method is called at the end of each real batch.

`public abstract void addEffectiveBatchObs (int s, int h, double l)`

Adds an observation to each statistical collector corresponding to an effective batch. The effective batch for which this method is called has length `l`, and regroups real batches `s`, ..., `s + h - 1`. This method is called after each error check if aggregation is turned ON, or after each real batch if it is turned OFF.

`public int getRequiredNewBatches()`

Computes the approximate number of required real batches to be simulated before the simulation can be stopped. The default implementation always returns 0, which stops the simulation after `getTargetBatches` real batches are obtained; sequential sampling is not used by default.

Note: if the method uses `getRequiredNewObservations` with a statistical probe containing one observation per effective batch, this gives the number of additional effective batches to simulate. This value should be multiplied with `getNumAggregates` to get the number of additional real batches.

`public void init()`

Initializes the simulator for a new experiment. This method, called by `simulate`, resets the counter for the number of batches, calls `simulator().init`, followed by `initSimulation`.

`public Event getEndSimEvent()`

Returns the event used to stop the simulation at the end of the warmup or batches.

`public void warmup()`

Performs a warmup by calling `warmup`. By default, this method calls `warmup` with the value returned by `getWarmupTime`, but one can override this method to simulate the warmup differently. If the duration of the warmup period is not fixed, one can call `warmup` with `Double.POSITIVE_INFINITY`; this prevents the method from scheduling the ending event, and let the simulator call `simulator().stop` at appropriate time.

`public void warmup (double warmupTime)`

Performs a warmup of fixed duration `warmupTime`. This method simulates for `warmupTime` simulation time units, and initializes statistical probes for real batches through `initReal-BatchProbes`. If the duration of the warmup period is not fixed, one can call `warmup` with `Double.POSITIVE_INFINITY`; this prevents the method from scheduling the ending event, and let the simulator call `simulator().stop` at appropriate time.

`public void simulateBatch()`

Simulate a new batch with default length. By default, this method multiplies the current batch size with `getBatchSizeMultiplier`, and schedules the next end-batch event to happen in `getBatchSize*getBatchFraction` simulation time units, by using `simulateBatch`. After the batch is simulated, the batch-size multiplier and fraction are reset to 1. If the batch lengths are not fixed, one can override this method to call `simulateBatch` with `Double.POSITIVE_INFINITY`; this prevents the method from scheduling the ending event, and let the simulator call `simulator().stop` at appropriate time.

`public void simulateBatch (double batchLength)`

Simulates a batch with length `batchLength`. This method initializes the model-specific counters by using `initBatchStat`, simulates the batch, and adds observations using `addRealBatchObs`. It also calls `addEffectiveBatchObs` if aggregation is turned OFF. If the batch lengths are not fixed, one can call this method to call `simulateBatch` with `Double.POSITIVE_INFINITY`; this prevents the method from scheduling the ending event, and let the simulator call `simulator().stop` at appropriate time.

`public void adjustTargetBatches (int numNewBatches)`

Adjusts the target number of real batches to simulate `numNewBatches` additionnal real batches. This method clamps the target number of real batches to the maximal number of batches, and ensures that the target number of batches is a multiple of the minimal number of batches when aggregation is turned ON.

`public void simulateBatches ()`

Simulates batches until the number of completed real batches corresponds to the target number of batches. This method first allocates the capacity for simulating `getTargetBatches`. It then simulate each batch using `simulateBatch`. If aggregation is turned ON, this method also calls `initEffectiveBatchProbes`, and `addEffectiveBatchObs` to manage effective batches.

`public void simulate()`

Performs a batch means simulation. This method resets the state of the system by calling `init`, and calls `warmup` to perform the warmup. Then, the method calls `simulateBatches` and `getRequiredNewBatches` until the number of completed real batches equals or exceeds the target number of batches.

`public static double getSum (double[] a, int start, int length)`

Returns the sum of elements `start`, ..., `start + length - 1`, in the array `a`.

`public static double getSum (DoubleArrayList l, int start, int length)`

Returns the sum of elements `start`, ..., `start + length - 1`, in the array list `l`.

`public static double getSum (DoubleMatrix1D m, int start, int length)`

Returns the sum of elements `start`, ..., `start + length - 1`, in the 1D matrix `m`.

```
public static double[] getSum (double[][] a, int startColumn,
                               int numColumns)
```

Returns an array containing the sum of columns `startColumn`, ..., `startColumn + numColumns - 1`, in the 2D matrix represented by the 2D array `a`. The given array is assumed to be rectangular, i.e., each of its array elements has the same length.

```
public static double[] getSum (DoubleMatrix2D m, int startColumn,
                               int numColumns)
```

Returns an array containing the sum of columns `startColumn`, ..., `startColumn + numColumns - 1`, in the 2D matrix `m`.

`public static void regroupElements (double[] a, int x)`

Regroups the elements in array `a` by summing each successive `x` values. When this method returns, element `i` of the given array corresponds to the sum of elements `ix`, ..., `ix + x - 1` in the original array. If the size of the array is not a multiple of `x`, the remaining elements are summed up and added into an extra element of the transformed array. Remaining elements of the transformed array are set to 0.

`public static void regroupElements (DoubleArrayList l, int x)`

Same as `regroupElements` for an array list. The size of the list is also divided by `x`.

`public static void regroupElements (DoubleMatrix1D mat, int x)`

Same as `regroupElements` for a 1D matrix.

`public static void regroupElements (DoubleMatrix2D mat, int x)`

Same as `regroupElements` for a 2D matrix. This method regroups columns and considers each row as an independent array.

References

[1] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, New York, NY, third edition, 2000.